


Zaawansowane projektowanie obiektowe


Wzorce projektowe cz. I

Prowadzący: Bartosz Walter

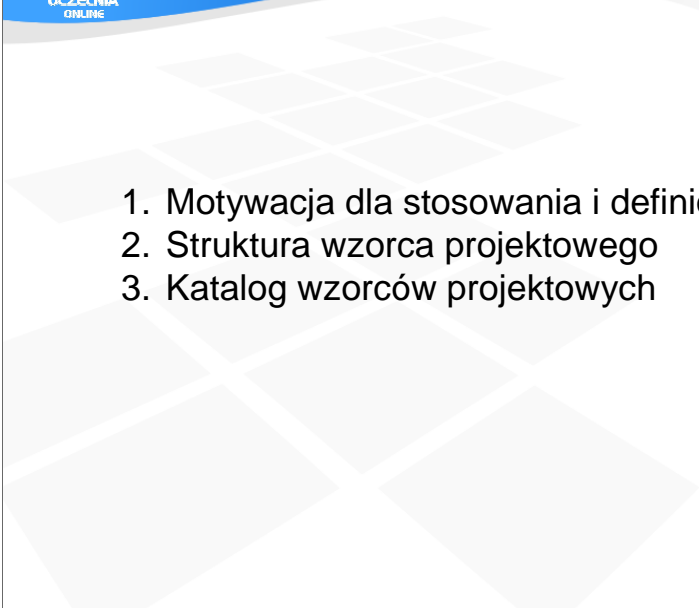


UCZELNIA  
ONLINE

Zaawansowane projektowanie obiektowe

Uczelnia  
ONLINE

Agenda



1. Motywacja dla stosowania i definiowania wzorców
2. Struktura wzorca projektowego
3. Katalog wzorców projektowych

Wzorce projektowe cz. I (2)

Wykład jest pierwszym z trzech poświęconych wzorcom projektowym. Podczas niego zostanie przedstawiona motywacja dla stosowania wzorców, typowy szablon wzorca oraz pierwsza część katalogu wzorców autorstwa tzw. Bandy Czterech.



Różne dziedziny inżynierii stawiają sobie podobne pytania:

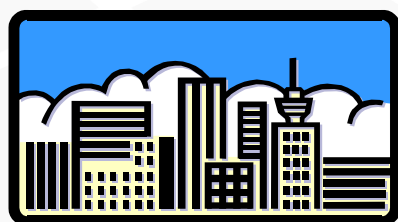
- Czy typowe problemy można rozwiązywać w powtarzalny sposób?
- Czy te problemy można przedstawić w sposób abstrakcyjny, tak aby były pomocne w tworzeniu rozwiązań w różnych konkretnych kontekstach?

Dążenia do jednolitości rozwiązań, ich klasyfikacji i uproszczenia, pojawiają w wielu dziedzinach inżynierii. Podstawowe pytanie dotyczy możliwości wielokrotnego wykorzystania raz sformułowanego rozwiązania danego problemu. Czy można zapisać to rozwiązanie w sposób ogólny, abstrahując od szczegółowych rozwiązań i jednocześnie umożliwiając wielokrotne jego wykorzystanie?



„Wzorzec opisuje **problem**, który powtarza się wielokrotnie **w danym środowisku**, oraz podaje **istotę jego rozwiązania** w taki sposób, aby można było je **zastosować miliony razy** bez potrzeby powtarzania tej samej pracy”

*Christopher Alexander „A pattern language”, 1977*



Pojęcie wzorca pojawiło się po raz pierwszy w architekturze. Jego twórcą był architekt, Christopher Alexander, który postawił pytanie, czy estetyka i funkcjonalność budowli i przestrzeni jest wartością obiektywną, wynikającą ze stosowania określonych rozwiązań, czy też każdorazowo zależy od pojedynczej koncepcji. Uznał, że wartości te można opisać za pomocą reguł, mówiących, że w celu osiągnięcia określonego celu należy zastosować pewne rozwiązanie, które pociąga za sobą określone konsekwencje.


Jest on także autorem pierwszej definicji wzorca, która jest na tyle ogólna, że można ją nadal stosować w oderwaniu od pierwotnej dziedziny zastosowań, czyli architektury. Mówi ona o problemie, kontekście, w jakim jest on osadzony, oraz szkieletcie rozwiązania opisanym ogólnie, na wysokim poziomie abstrakcji. Taki wzorzec, po nadaniu wartości zmiennym, jest gotowym rozwiązaniem znajdującym zastosowanie w konkretnej sytuacji.

Zaawansowane projektowanie obiektowe

UCZELNIA  
ONLINE

## Wzorce w budownictwie lądowym

Czy zbudować most, opierając przęsło na kolejnych filarach połączonych łukiem, tak aby łuk usztywniał przęsło, stanowiąc jego podparcie na całej długości przęsła, czy też mocując przęsło z obu stron za pomocą lin stalowych o kolejno coraz krótszych długościach do pylonów umieszczonych pośrodku długości mostu?



na podstawie przykładu R. Johnsona

Wzorce projektowe cz. I (5)

Aby przybliżyć pojęcie wzorca, przyjrzyjmy się dylematowi projektanta budowlanego, który opisuje alternatywne sposoby konstrukcji mostu. Z każdym rozwiązaniem związane są pewne wymagania wstępne, uwarunkowania konstrukcyjne i konsekwencje. Wyrażenie ich w sposób opisowy jest możliwe, ale dość skomplikowane i narażone na pomyłki. Trzeba bowiem niejako na nowo przemyśleć poszczególne elementy projektu, uwzględnić zadania, jakie stoją przed projektowaną budowlą, warunki klimatyczne etc.

Zaawansowane projektowanie obiektowe

UCZELNIA ONLINE

## Wzorce w budownictwie lądowym

Czy zbudować most łukowy czy podwieszany?



na podstawie przykładu R. Johnsona

Wzorce projektowe cz. I (6)

Dlatego łatwiejsze jest posłużenie się wzorcem, w którym zawarte są gotowe informacje o możliwości zastosowania go w konkretnej sytuacji i efektach takiego rozwiązania. Mosty łukowe i mosty podwieszane wymagają innego rodzaju podparcia, pozwalają na osiągnięcie innej długości przęsła oraz inaczej rozkładają działające siły. W zależności od miejscowych warunków, szerokości rzeki i innych czynników można dokonać wyboru między tymi konkurencyjnymi rozwiązaniami.



### Wzorce w inżynierii oprogramowania

- **wzorce architektoniczne** – poziom integracji komponentów
- **wzorce projektowe** – poziom interakcji między klasami
- **wzorce analityczne** – poziom opisu rzeczywistości
- **wzorce implementacyjne** – poziom języka programowania

Wzorzec projektowy identyfikuje i opisuje pewną abstrakcję, której poziom znajduje się powyżej poziomu abstrakcji pojedynczej klasy, instancji lub komponentu.

*E. Gamma, R. Johnson, R. Helm, J. Vlissides, 1994*

Wzorce projektowe stanowiły pierwszy objaw „wzorcomanii” w inżynierii oprogramowania. Dążenie do półformalnego opisu wiedzy na temat „dobrych rozwiązań” znajduje coraz szerszy oddźwięk w społeczności badaczy i praktyków wytwarzania oprogramowania. Obecnie pojęcie wzorca jest często wykorzystywane w wielu innych zastosowaniach, także na poziomie architektury, testowania, analizy i implementacji oprogramowania.



- **Wzorce kreacyjne**
  - abstrakcyjne metody tworzenia obiektów
  - uniezależnienie systemu od sposobu tworzenia obiektów
- **Wzorce strukturalne**
  - sposób wiązania obiektów w struktury
  - właściwe wykorzystanie dziedziczenia i kompozycji
- **Wzorce behawioralne**
  - algorytmy i przydział odpowiedzialności
  - opis przepływu kontroli i interakcji

Pierwszą szeroko znaną publikacją na temat wzorców była książka autorstwa E. Gammy, R. Helma, R. Johnsona i J. Vlissidesa, znanych także jako Banda Czterech (ang. *Gang of Four*).

Autorzy książki zaproponowali podstawowy podział wzorców na trzy kategorie: wzorce kreacyjne (ang. *creational*), dotyczące tworzenia obiektów lub struktur obiektowych, wzorce strukturalne (ang. *structural*), opisujące sposób wiązania obiektów w złożone struktury o określonych właściwościach, oraz wzorce behawioralne (ang. *behavioral*), opisujące algorytmy realizacji typowych zadań.





Wzorzec projektowy jest opisany przez:

- **nazwę** – lakoniczny opis istoty wzorca
- **klasyfikację** – kategorię, do której wzorzec należy
- **cel** – do czego wzorzec służy
- **aliasy** – inne nazwy, pod którymi jest znany
- **motywację** – scenariusz opisujący problem i rozwiązanie
- **zastosowania** – sytuacje, w których wzorzec jest stosowany
- **strukturę** – graficzną reprezentację klas składowych wzorca

Każdy wzorzec należący do katalogu zaproponowanego przez „Bandę Czterech” opisany jest przez zestaw atrybutów, dzięki którym jego właściwości są przedstawione w usystematyzowany, powtarzalny i obiektywny sposób. W ten sposób powstał szablon wzorca projektowego.

Podczas wykładu jednak każdy wzorzec zostanie opisany tylko przez część atrybutów, w zakresie pozwalającym poznać przeznaczenie wzorca i istotę jego konstrukcji. Szczegółowego opisu można szukać w literaturze.

Nazwa wzorca jest dobrana tak, aby szybko nasuwać skojarzenia z przeznaczeniem wzorca. Nazwy pierwotnie zostały sformułowane po angielsku, i tak też będą używane w trakcie wykładu. Stosowanie spójnego, anglojęzycznego nazewnictwa pozwala na łatwą komunikację, dlatego unikanie polskich tłumaczeń wydaje się uzasadnione.

Cel wzorca krótko opisuje kontekst, w jakim go warto zastosować, i jakie efekty można przy jego pomocy osiągnąć.

Bardzo ważnym elementem jest opis struktury wzorca, przede wszystkim w zakresie powiązań pomiędzy uczestniczącymi w nim klasami w postaci diagramu klas UML. Aspekt dynamiczny opisywany jest w atrybucie dotyczącym kolaboracji.



- **uczestników** – nazwy i odpowiedzialności klas składowych wzorca
- **współdziałania** – opis współpracy między uczestnikami
- **konsekwencje** – efekty zastosowania wzorca
- **implementację** – opis implementacji wzorca w danym języku
- **przykład** – kod stosujący wzorzec
- **pokrewne wzorce** – wzorce używane w podobnym kontekście

Lista uczestników wzorca zawiera nie tylko nazwy ról klas wchodzących w jego skład, ale także zakres ich odpowiedzialności. Jest to uszczegółowienie informacji, które znajdują się na diagramie struktury.

Często pomijaną składową każdego wzorca jest informacja o konsekwencjach, jakie niesie jego zastosowanie, szczególnie negatywnych. Wykorzystanie wzorca często narzuca pewne decyzje, dlatego projektant powinien być świadomy ich związków z tym wzorcem.

Przykład pozwala lepiej zrozumieć charakter, przeznaczenie i strukturę wzorca.



- Katalog wzorców projektowych *Gang of Four* (Gamma, Johnson, Helm, Vlissides) obejmuje 23 wzorce:
  - **kreacyjne**: *Abstract Factory, Builder, Factory Method, Prototype, Singleton*
  - **strukturalne**: *Adapter, Bridge, Composite, Decorator, Composite, Facade, Proxy, Flyweight*
  - **behawioralne**: *Chain of Responsibility, Command, Interpreter, Mediator, Iterator, Memento, Observer, State, Strategy, Template Method, Visitor*
- Lista wzorców jest sukcesywnie uzupełniana przez innych autorów

Katalog przedstawiony w książce Bandy czterech składa się z 24 wzorców, z których 5 należy do kategorii wzorców kreacyjnych, 8 – strukturalnych, a 11 – behawioralnych.

Podczas wykładu zostanie przedstawionych 23 wzorce należących do kanonicznego katalogu (pominięty zostanie wzorzec Interpreter, z uwagi na ograniczone zastosowania). Dodatkowo zostanie omówiony nie należący kanonu wzorzec puli obiektów.



- Zapewnienie, że klasa posiada jedną instancję wewnątrz całej aplikacji
- Stworzenie punktu dostępowego do tej instancji

*Gang of Four*

Wzorce projektowe cz. I (12)

Singleton jest najprostszym wzorcem projektowym. Jego celem jest stworzenie obiektowej alternatywy dla zmiennych globalnych, nieobecnych w wielu językach obiektowych: zapewnienie istnienia w aplikacji tylko jednej instancji danej klasy oraz udostępnienie tej instancji w łatwo dostępny i intuicyjny sposób, zwykle poprzez dedykowaną metodę statyczną.

Zaawansowane projektowanie obiektowe

Uczelnia ONLINE

## Singleton: struktura i uczestnicy

```

classDiagram
    class Singleton {
        -instance
        -singletonData
        +getInstance()
        +singletonOperation()
        +getSingletonData()
        +getSingleton()
        +Singleton()
    }
    Singleton --> Singleton : return instance;
  
```

### Singleton

- definiuje statyczną metodę *getInstance()* udostępniającą instancję klasy
- ogranicza dostęp do konstruktora do własnej klasy i podklas
- jest odpowiedzialny za tworzenie instancji własnej klasy

Wzorce projektowe cz. I (13)

Singleton składa się z jednej klasy, która zarządza swoją własną jedyną instancją. Instancja jest przechowywana w postaci prywatnego pola statycznego, natomiast zarządzaniem nią zajmuje się publiczna metoda statyczna o nazwie *getInstance()*. Postępuje ona według następującego algorytmu: jeżeli pole statyczne przechowujące instancję klasy ma wartość *null* (czyli instancja dotąd nie została utworzona), wówczas instancja taka jest tworzona i zapamiętywana w tym polu. Dzięki temu, niezależnie od tego, który raz wywoływana jest metoda, zawsze zwraca ona utworzoną i jedyną instancję klasy.

Aby uniemożliwić klientom samodzielne tworzenie instancji z pominięciem metody statycznej, klasa Singleton uniemożliwia dostęp do konstruktora z zewnątrz, zwykle czyniąc go prywatnym lub chronionym.



- Singleton przejmuje odpowiedzialność za tworzenie instancji własnej klasy
- Klient nie zarządza instancją klasy; otrzymuje ją na żądanie
- Singleton może zarządzać także swoimi podklasami
- Singleton można łatwo rozszerzyć do puli obiektów
- Singleton jest zwykle obiektem bezstanowym
- Singleton zachowuje się podobnie do zmiennej globalnej
- Singleton może powodować zwiększenie liczby powiązań w systemie

Singleton jest przede wszystkim obiekowym sposobem na zapewnienie, że zostanie utworzona dokładnie jedna instancja klasy, która będzie dostępna dla wszystkich obiektów aplikacji. Warto zauważyć, że ten wzorec pozwala także przenieść odpowiedzialność za tworzenie obiektu z klienta na dedykowaną metodę. Koncepcja ta zostanie dalej rozwinięta we wzorcach Factory Method i Abstract Factory.

Singleton jest zwykle obiektem bezstanowym, tzn. sposób działania metody statycznej nie zależy od stanu, w jakim znajduje się program: klient otrzymuje instancję klasy na żądanie, niezależnie od tego, czy została ona utworzona wcześniej, czy nie. Singleton pozwala także stosować dziedziczenie w celu zmiany przez siebie tworzonej klasy i zwracać także instancje podklas. Dołączenie podklasy do wzorca nie wymaga modyfikacji po stronie klienta.

Singleton w pewnym sensie może także być uważany za szczególny przypadek obiektu Pool of Objects; może także być stosunkowo łatwo rozszerzony do takiej postaci.



```
static public Tax getInstance() {  
    if (instance == null) {  
        synchronize (this) {  
            if (instance == null) {  
                instance == new TaxA();  
            }  
        }  
    }  
    return instance;  
}
```

Istnienie obiektu *instance* jest sprawdzane dwukrotnie, na zewnątrz i wewnątrz bloku synchronizacji

*Shalloway & Trott (2001)*

Wzorce projektowe cz. I (15)

W języku Java implementacja tego wzorca napotyka na wiele trudności ze względu na sposób wykonywania programów i konstrukcję maszyny wirtualnej, w której są uruchamiane programy. M.in. w programie wielowątkowym istnieje możliwość, że wskutek przerwania wykonywania metody w momencie sprawdzania, czy instancja obiektu została już utworzona, kontrolę przejmie drugi wątek, który utworzy swoją własną instancję.

W celu rozwiązania tego problemu można zastosować zmodyfikowaną wersję algorytmu blokowania dwufazowego (2PL). Zakłada ona, że istnienie instancji obiektu jest sprawdzane dwukrotnie: na zewnątrz i wewnątrz bloku synchronizacji, w którym instancja ta jest tworzona. Taka konstrukcja, mimo pewnego narzutu związanego z synchronizacją wątków, pozwala uniknąć utworzenia wielu instancji klasy.



## Singleton: implementacja z *class loaderami*

```
public class TaxA extends Tax {
    private static class Instance {
        static final Tax instance = new TaxA();
    }

    private TaxA() {}
    public static Tax getInstance() {
        return Instance.instance;
    }
}
```

*Class loader* ładuje pojedynczą klasę *TaxA.Instance*, która przechowuje pojedynczą instancję klasy *Tax*

*Shalloway & Trott (2001)*

Wzorce projektowe cz. I (16)


Inne rozwiązania wykorzystuje mechanizm działania tzw. class loader'ów wewnątrz maszyny wirtualnej. Obiekty class loader służą do ładowania klas i są zorganizowane w postaci drzewa. Każdy z nich, otrzymując żądanie załadowania klasy, aby uniknąć wielokrotnego załadowania tej samej klasy, zawsze najpierw konsultuje się ze swoim nadrzędnym class loaderem, czy nie załadował on już poszukiwanej klasy. W ten sposób poprawnie napisane class loadery (mogą one być definiowane przez programistę) zapewniają, że w maszynie wirtualnej zawsze znajduje się co najwyżej jedna reprezentacja danej klasy.

Wzorec może być wówczas zaimplementowany w postaci instancji klasy *TaxA* w statycznej klasie wewnętrznej *Instance*. Instancja ta jest tworzona w momencie załadowania klasy *TaxA* (oraz *Instance*) do maszyny wirtualnej, a sposób działania obiektu class loader zapewnia, że nie zostanie utworzona więcej niż jedna jej instancja.

Rozwiązanie to działa poprawnie, o ile obiekty class loader zdefiniowane przez programistę zachowują się poprawnie, tj. konsultują ładowanie każdej klasy ze swoim nadrzędnym class loaderem. Jeżeli ta zasada zostanie naruszona, wówczas nadal istnieje niebezpieczeństwo utworzenia wielu instancji.



Zaawansowane projektowanie obiektowe

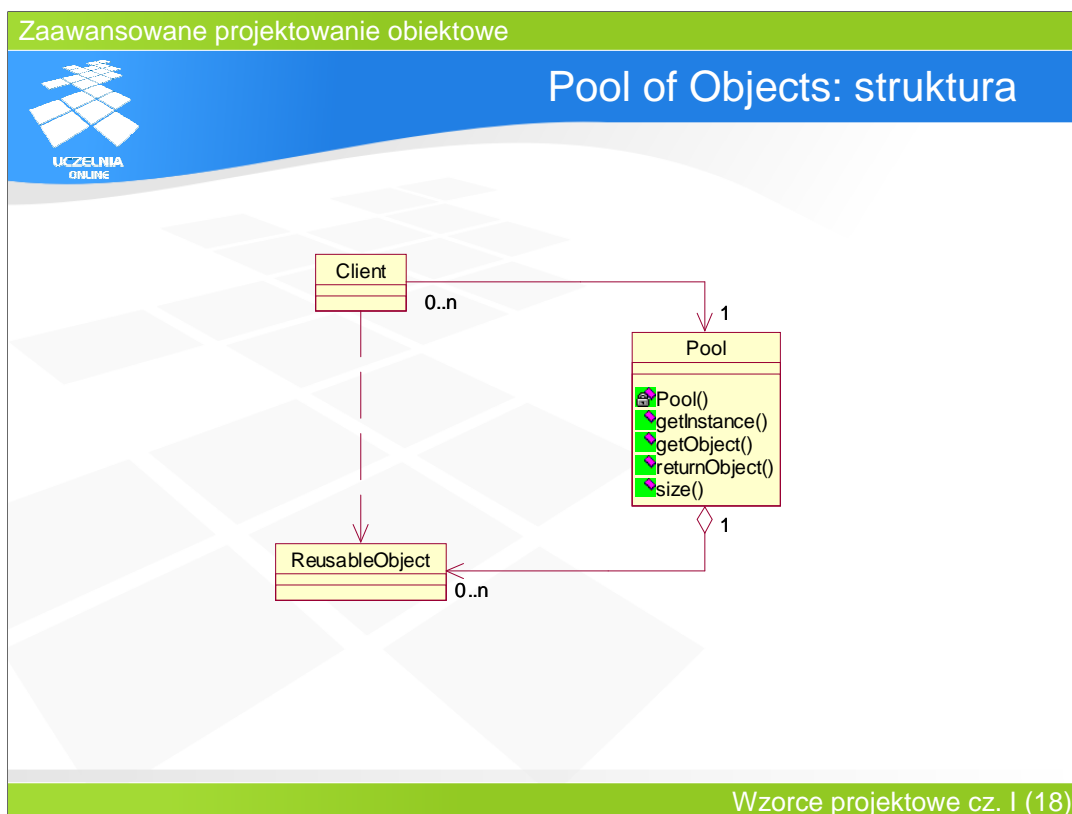
Pool of Objects: cel

- Zarządzanie grupą obiektów reprezentujących zasoby wielokrotnego użycia
- Ograniczenie kosztów tworzenia i usuwania obiektów

*Shalloway & Trott (2001)*

Wzorce projektowe cz. I (17)

Pula obiektów stanowi pewnego rodzaju rozszerzenie idei wzorca Singleton oraz opisanego dalej wzorca Factory Method: pozwala na przesunięcie odpowiedzialności za tworzenie produktów na oddzielny obiekt, a jednocześnie umożliwia wielokrotne wykorzystanie poszczególnych instancji obiektów. Ma to szczególne znaczenie w przypadku produktów reprezentujących zasoby, które są czasowo alokowane na rzecz konkretnego klienta. Pozwala to istotnie ograniczyć koszt związany z tworzeniem i usuwaniem obiektów.



Najważniejszym elementem wzorca jest klasa `Pool`, która w porównaniu do wymienionych wcześniej wzorców `Singleton` i `FactoryMethod` ma zwiększony zakres odpowiedzialności. Nie tylko zajmuje się tworzeniem instancji klasy `ReusableObject`, ale także zarządzaniem cyklem życia już utworzonych obiektów. Najczęściej klasa ta utrzymuje zbiór aktywnych obiektów `ReusableObject`, które są przekazywane klientom na żądanie i przyjmowane od nich z powrotem po wykorzystaniu. Zatem klasa `Pool` posiada interfejs służący do tworzenia produktu (metoda `getInstance()`) oraz ich zwracania (metoda `returnInstance()`). Z punktu widzenia klienta obiekt klasy `Pool` jest fabryką produktów, ponieważ klient nie musi zajmować się ich tworzeniem, zarządzaniem, odtwarzaniem etc.



- **Pool**
  - definiuje punkt dostępu do obiektów *Reusable Object*
  - zarządza cyklem życia obiektów *Reusable Object*
- **Reusable Object**
  - definiuje swój cykl życia
  - może być powtórnie wykorzystany
- **Client**
  - otrzymuje obiekty *Reusable Object* za pośrednictwem obiektu *Pool*

Najważniejsze dwie funkcje obiektu Pool to zdefiniowanie punktu dostępu (zarówno tworzenia, jak i zwrotu) do obiektów typu ReusableObject, oraz zarządzanie cyklem ich życia. Cykl życia produktu składa się zwykle z fazy inicjalizacji, obsługi i finalizacji. Ponieważ klient oczekuje produktu gotowego do natychmiastowego użytku, dlatego fazy inicjalizacji i finalizacji są pod kontrolą obiektu Pool.

Obiekt ReusableObject musi posiadać zdefiniowany cykl życia: zestaw metod odpowiednio modyfikujących jego stan. Najważniejszą cechą tego obiektu jest możliwość jego ponownego użycia przez innego klienta.

Klient żąda obiektu ReusableObject za pomocą obiektu Pool i w ten sam sposób zwalnia przydzielony obiekt.




- **Zwiększona wydajność**
  - obiekty *ReusableObject* są tworzone w ograniczonej liczbie instancji i wykorzystywane wielokrotnie
  - zrównoważone obciążenie zasobów
- **Lepsza hermetyzacja**
  - klient nie zajmuje się tworzeniem i obsługą obiektów *ReusableObject*

Dzięki wykorzystaniu wzorca Pool of Objects, obiekty *ReusableObject* są tworzone w ograniczonej liczbie instancji i mogą być następnie wielokrotnie wykorzystywane. Pozwala to usunąć istotny koszt związany z tworzeniem obiektów. Jest on szczególnie dokuczliwy, gdy liczba żądań jest duża, a czas wykorzystania obiektu bardzo krótki, np. w kontenerach Java Servlets obsługujących żądania HTTP. Do każdego żądania jest przydzielana para obiektów reprezentujących żądanie i odpowiedź HTTP. Skalowalność wymaga, aby liczba jednoczesnych żądań wynosiła przynajmniej kilkadziesiąt, czego nie dałoby się osiągnąć bez efektywnego mechanizmu zarządzania pulą obiektów.

Innym, często spotykanym przykładem, jest dostęp do bazy danych za pomocą interfejsów JDBC. Za każdym razem wymagane jest udostępnienie klientowi obiektu typu *Connection*, które na czas operacji na bazie danych musi być związane z jednym wątkiem. Utworzenie obiektu *Connection* jest bardzo czasochłonne, dlatego zwykle jest on umieszczany w puli, tak aby po jego wykorzystaniu przez jeden wątek mógł on trafić do niej z powrotem. Liczba jednocześnie istniejących obiektów jest konfigurowalna, tak aby zapewnić obsługę wszystkich żądań. Obiekt Pool może także wykorzystywać skomplikowane algorytmy heurystyczne w celu przewidywania zapotrzebowania na obiekty *ReusableObject* i dostosowywania do potrzeb liczby obiektów przechowywanych w puli.

Ponadto wzorec ten poprawia hermetyzację obiektu *ReusableObject*: klient nie zajmuje się ich obsługą, a jedynie korzysta z oferowanych przez nie usług.

Zaawansowane projektowanie obiektowe



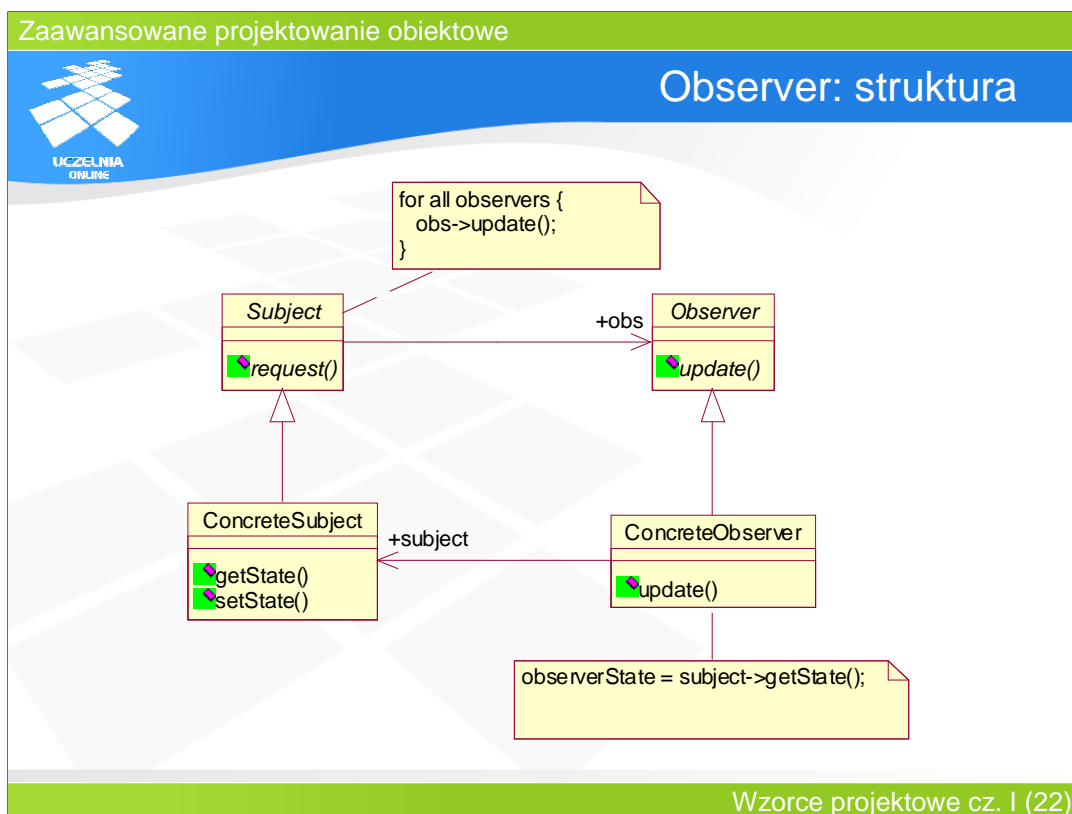
Observer: cel

- Utworzenie zależności typu jeden-wiele pomiędzy obiektami
- Informacja o zmianie stanu wyróżnionego obiektu jest przekazywana wszystkim pozostałym obiektom

*Gang of Four*

Wzorce projektowe cz. I (21)

Wzorzec Observer służy do stworzenia relacji typu jeden-wiele łączącej grupę obiektów. Dzięki niemu zmiana stanu obiektu po stronie „jeden” umożliwi automatyczne powiadomienie o niej wszystkich innych zainteresowanych obiektów (tzw. obserwatorów).



Wzorzec składa się z dwóch ról: obiektu obserwowanego (**Subject**) oraz obserwatorów (**Observer**). Obiekt **Subject** posiada metody pozwalające na dołączanie i odłączanie obserwatorów: każdy zainteresowany obiekt może się zarejestrować jako obserwator. Ponadto posiada metodę *notify()*, służącą do powiadamiania wszystkich zarejestrowanych obserwatorów poprzez wywołanie w pętli na ich rzecz metody *update()*.

Interfejs **Observer** jest bardzo prosty i zawiera tylko jedną metodę – *update()*. Metoda ta jest wykorzystywana właśnie do powiadamiania obiektu o zmianie stanu obiektu obserwowanego, a sam interfejs jest jedyną informacją, jaką o obserwatorach posiada ten obiekt.



- **Subject**
  - utrzymuje rejestr obiektów *Observer*
  - umożliwia dołączanie i odłączanie obiektów *Observer*
- **Observer**
  - udostępnia interfejs do powiadamiania o zmianach
- **Concrete Subject**
  - przechowuje stan istotny dla obiektów *Concrete Observer*
  - powiadamia obiekty *Concrete Observer*
- **Concrete Observer**
  - aktualizuje swój stan na podstawie powiadomienia

W ramach wymienionych dwóch podstawowych ról: obserwatora i obiektu obserwowanego, można wydzielić dodatkowo warstwę abstrakcji i warstwę implementacji. W tej pierwszej znajdują się interfejsy *Subject* i *Observer*, które definiują zakres funkcjonalności poszczególnych klas, oraz klasy *ConcreteSubject* i *ConcreteObserver*, które są przykładami realizacji tych kontraktów.

W języku Java rola obiektu obserwowanego jest reprezentowana przez klasę *java.util.Observable*, natomiast obserwatory implementują interfejs *java.util.Observer*. Dzięki temu implementacja wzorca w tym języku jest znacznie uproszczonym zadaniem.



- Luźniejsze powiązania pomiędzy obiektami:
  - obiekt *Subject* komunikuje się z innymi obiektami przez interfejs *Observer*
  - obiekty *Subject* i *Observers* mogą należeć do różnych warstw abstrakcji
- Programowe rozgłaszanie komunikatów
- Spójność stanu pomiędzy obiektami *Subject* i *Observers*
- Skalowalność aktualizacji
  - *push*: *Observers* otrzymują kompletny stan obiektu *Subject*
  - *pull*: *Observers* otrzymują powiadomienie i referencję do obiektu *Subject*


Wzorzec Observer pozwala na znaczne ograniczenie powiązań i zależności pomiędzy obserwatorami i obiektem obserwowanym. Wprawdzie obiekt obserwowany posiada referencje do obserwatorów, jednak wiedza jest ograniczona tylko do znajomości interfejsu *Observer*. Także obserwatory nie muszą znać obiektu *Subject* w momencie wywołania ich metody *update()*, ponieważ otrzymują powiadomienia asynchroniczne.

Dzięki ogólności interfejsu *Observer* obiekty uczestniczące we wzorcu mogą należeć do różnych warstw abstrakcji. Wzorzec pozwala zachować spójność pomiędzy warstwami aplikacji, ponieważ informacje o zmianach w jednej warstwie są przekazywane natychmiast do pozostałych obiektów. Jest to szczególnie często wykorzystywane do komunikacji w wielu systemach okienkowych. Zamiennie zamiast nazwy *Observer* wykorzystuje się nazwę *Listener*.

Ponieważ ilość informacji przekazywanych obiektom *Observer* może istotnie wpływać na wydajność systemu, dlatego istnieją dwa podejścia do implementacji tego wzorca. W modelu *push* każdy obserwator otrzymuje w postaci parametru metody *update()* pełną informację o stanie obiektu *Subject*. W modelu *pull* obserwatory otrzymują tylko referencję do obiektu *Subject*, dzięki której mogą następnie odpytać go o szczegóły dotyczące zmiany. Ten ostatni model jest zatem znacznie lepiej skalowalny, szczególnie w przypadku wywoływania tych metod w środowisku rozproszonym.



Zaawansowane projektowanie obiektowe

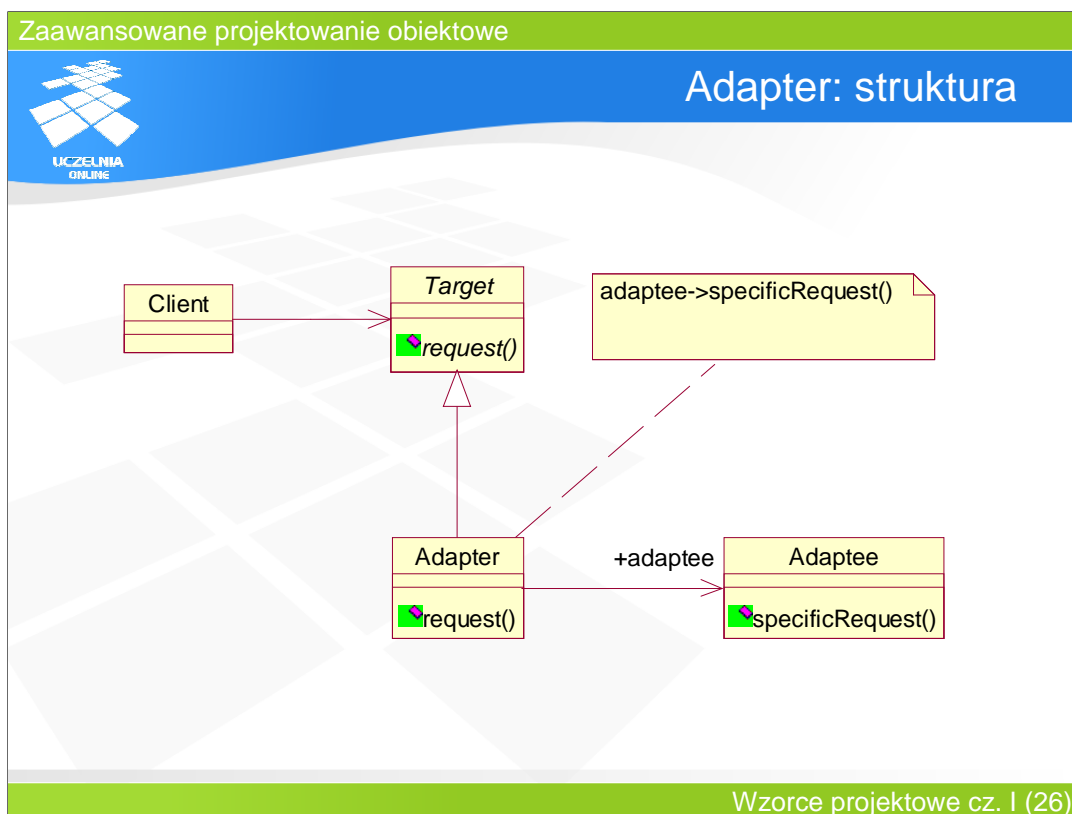
 Adapter: cel

- Umożliwienie współpracy obiektów o niezgodnych typach
- Tłumaczenie protokołów obiektowych

*Gang of Four*

Wzorce projektowe cz. I (25)

Adapter (znany także pod nazwą Wrapper) służy do adaptacji interfejsów obiektowych, tak aby możliwa była współpraca obiektów o niezgodnych typach. Szczególnie istotną rolę odgrywa on w przypadku wykorzystania gotowych bibliotek o interfejsach niezgodnych ze stosowanymi w aplikacji.




Struktura wzorca składa się z trzech podstawowych klas: Target, Adaptee oraz Adapter. Target jest interfejsem, którego oczekuje klient. Obiektem dostarczającym żądanej przez klienta funkcjonalności, ale niezgodnego pod względem typu, jest Adaptee. Rolą Adaptera, który implementuje typ Target, jest przetłumaczenie wywołania metody należącej do typu Target poprzez wykonanie innej metody (lub grupy metod) w klasie Adaptee. Dzięki temu klient współpracuje z obiektem Adapter o akceptowanym przez siebie interfejsie Target, jednocześnie wykorzystując funkcjonalność dostarczoną przez Adaptee.

Alternatywna nazwa wzorca – Wrapper, która oznacza opakowanie, bardzo dobrze opisuje rolę obiektu Adapter: pełnić wobec Klienta rolę otoczki, która umożliwi przetłumaczenie jego żądań na protokół zrozumiały dla faktycznego wykonawcy poleceń.

Wzorec ten posiada także wersję wykorzystującą dziedziczenie w relacji Adapter-Adaptee. Jednak wersja ta ma pewne niedogodności: powiązania między obiektami są ustalane w momencie kompilacji i nie mogą ulec zmianie; ponadto, język programowania musi umożliwiać stosowanie wielokrotnego dziedziczenia lub dziedziczenia i implementacji interfejsu (jak w przypadku języków Java i C#).

Zaawansowane projektowanie obiektowe



Adapter: uczestnicy

- **Target**
  - definiuje interfejs specyficzny dla klienta
- **Client**
  - współpracuje z obiektami typu *Target*
- **Adaptee**
  - posiada interfejs wymagający adaptacji
- **Adapter**
  - adaptuje interfejs *Adaptee* do interfejsu *Target*

Wzorce projektowe cz. I (27)

We wzorcu występują trzy podstawowe obiekty: Target, definiujący interfejs wymagany przez klienta, i poprzez który chce on wykorzystywać określoną funkcjonalność, Adaptee, który posiada tę funkcjonalność, ale jest niezgodny pod względem typu z interfejsem Target, oraz Adapter, dokonujący translacji pomiędzy nimi.




- **Duża elastyczność**
  - pojedynczy *Adapter* może współpracować z wieloma obiektami *Adaptee* naraz
  - *Adapter* może dodawać funkcjonalność do *Adaptee* (zob. wzorzec *Decorator*)
- **Utrudnione pokrywanie metod *Adaptora***
  - konieczne utworzenie podklas obiektu *Adaptee* i bezpośrednie odwołania do nich
- **Kompozycja i dziedziczenie jako mechanizmy adaptacji**

Adapter, niezależnie od swojego podstawowego przeznaczenia, wprowadza dodatkową warstwę abstrakcji, która pozwala uniknąć bezpośredniej zależności pomiędzy klientem a obiektem wykonującym żądania. Dzięki temu relację pomiędzy nimi można traktować w sposób elastyczny, np. zmieniając liczbę aktywnych obiektów *Adaptee*, którymi zarządza jeden *Adapter*.

Wzorzec może alternatywnie wykorzystywać dwa rodzaje relacji: kompozycję i dziedziczenie; użycie tej pierwszej daje więcej możliwości modyfikacji systemu w przyszłości.

Możliwa jest również rozbudowa tego wzorca do wzorca *Decorator*, tzn. rozszerzenie funkcjonalności obiektu *Adaptee* w *Adapterze*.

Zaawansowane projektowanie obiektowe

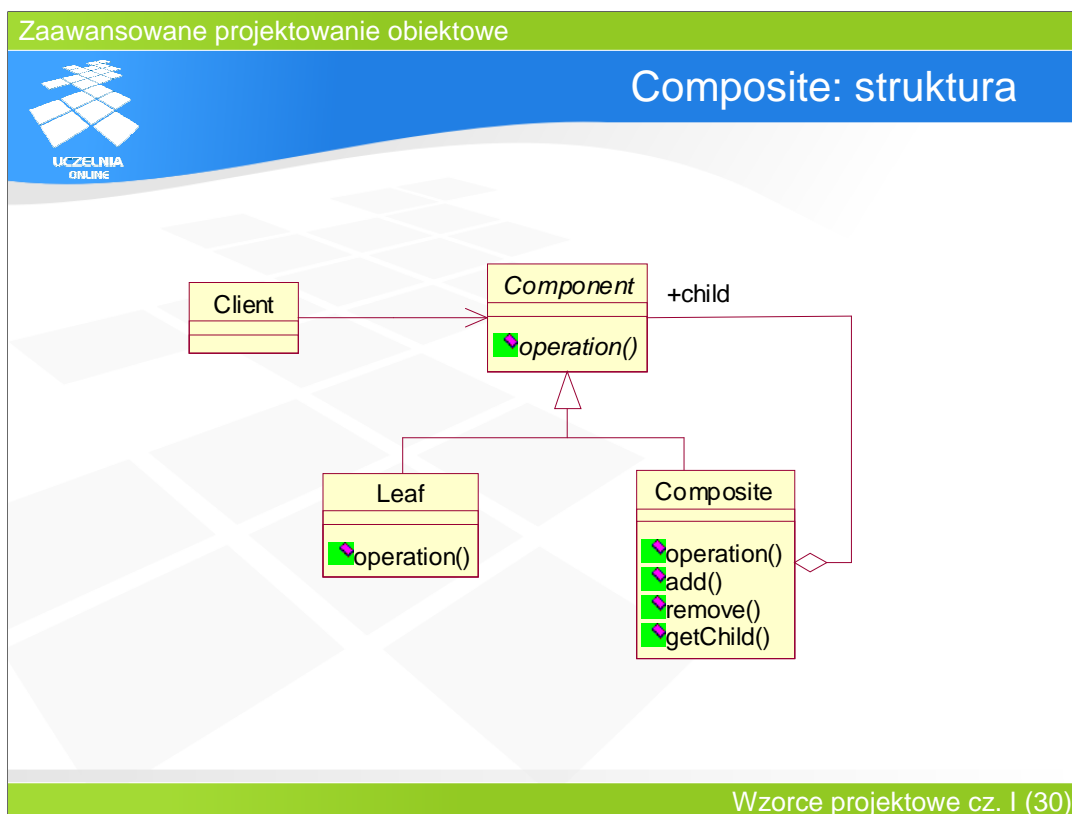
Composite: cel

- Organizowanie obiektów w struktury drzewiaste reprezentujące relacje typu **całość-część**
- **Jednolita obsługa** pojedynczych obiektów i złożonych struktur

Gang of Four

Wzorce projektowe cz. I (29)

Composite jest bardzo często stosowanym wzorcem służącym do reprezentacji struktur drzewiastych typu całość-część tak, aby sposób zarządzania strukturą nie zależał od jej złożoności. Jest często stosowany w obiektowych bibliotekach okienkowych jako metoda zarządzania widokami zbudowanymi z wielu widget'ów.




Centralnym elementem wzorca jest interfejs `Component`, który reprezentuje dowolny obiekt w strukturze drzewiastej. Posiada on możliwości dodawania i usuwania swojego obiektu potomnego (oczywiście, także typu `Component`) oraz odwołania się do wybranego potomka. Zawiera on także metodę `operation()`, którą należy wykonać na każdym węźle struktury.

Interfejs `Component` posiada dwie implementacje: `Leaf` oraz `Composite`. Klasa `Leaf` reprezentuje obiekty, które nie posiadają potomków (czyli liście w strukturze), natomiast `Composite` jest dowolnym węzłem pośrednim. Ponieważ każdy węzeł pośredni zarządza także poddrzewem, którego jest korzeniem, dlatego metoda `operation()`, poza wykonaniem operacji specyficznych dla każdego węzła, wywołuje swoje odpowiedniki w obiektach potomnych, w ten sposób propagując wywołanie.

Z punktu widzenia klienta taka struktura umożliwia zarządzanie całością za pomocą jednego obiektu – korzenia drzewa. Niepotrzebna jest także wiedza o rozmiarze drzewa, ponieważ wywołanie zostanie przekazane automatycznie do wszystkich jego elementów.

Zaawansowane projektowanie obiektowe


Composite: uczestnicy

- **Component**
  - deklaruje wspólny interfejs dla obiektów znajdujących się strukturze
  - implementuje wspólną funkcjonalność wszystkich obiektów
- **Leaf**
  - reprezentuje węzeł bez potomków
- **Composite**
  - reprezentuje węzeł z potomkami
  - przechowuje referencje do potomków
  - deleguje otrzymane polecenia do potomków

Wzorce projektowe cz. I (31)

Component, podstawowy element wzorca, przede wszystkim deklaruje wspólny interfejs dla wszystkich obiektów. Jego implementacje, Leaf i Composite, reprezentują odpowiednio węzły bez potomków i węzły pośrednie.

Zaawansowane projektowanie obiektowe



Composite: konsekwencje

- Elastyczna definicja struktur drzewiastych
- Proste dodawanie nowych komponentów
- Proste i spójne zarządzanie strukturą o dowolnej liczbie elementów


Wzorce projektowe cz. I (32)

Mechanizm ten jest jednym z najczęściej wykorzystywanych wzorców projektowych, np. w systemach okienkowych. Strukturę drzewiastą tworzą wówczas składowe okienek: przyciski, etykiety, listy etc.

Popularność tego wzorca wynika z elastycznego zarządzania złożonymi strukturami z punktu widzenia klienta: nie jest wymagana wiedza o rozmiarze i dokładnej strukturze drzewa. Ponadto wszystkie elementy struktury realizują ten sam algorytm, co znacznie ułatwia ich testowanie.



Zaawansowane projektowanie obiektowe

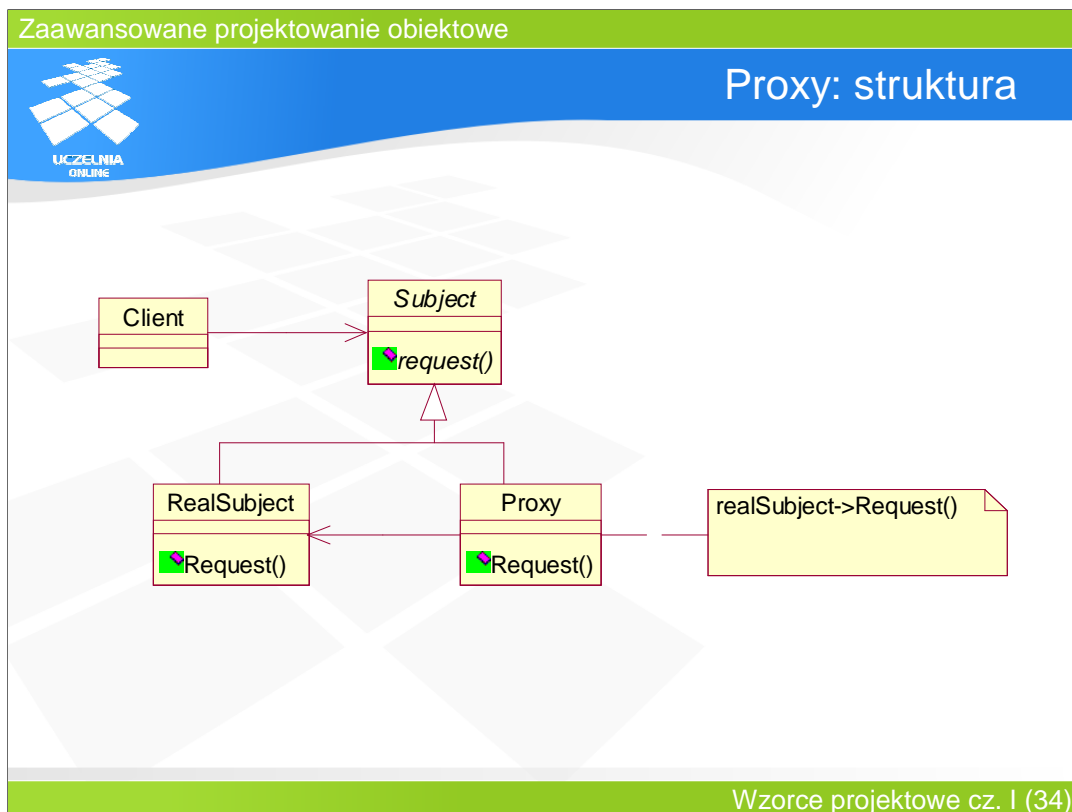
Proxy: cel

- Dostarczenie zamiennika dla obiektu w celu jego kontroli i ochrony
- Przezroczyste odsunięcie inicjalizacji obiektu w czasie

Gang of Four

Wzorce projektowe cz. I (33)


Celem wzorca Proxy jest zastąpienie obiektu docelowego tymczasowym substytutem, który może pełnić trzy funkcje: odsunąć w czasie moment utworzenia obiektu docelowego, będzie kontrolował do niego dostęp lub pozwoli odwoływać się do obiektu zdalnego. Z punktu widzenia klienta substytut powinien być przezroczysty i nie może mieć wpływu na sposób interakcji z obiektem docelowym.



Centralnym elementem wzorca jest interfejs **Subject**, który posiada wiele implementacji. Jedną z nich jest obiekt **RealSubject** – obiekt docelowy posiadający funkcjonalność wymaganą przez klienta. Drugą – obiekt proxy, który posiada referencję do obiektu **RealSubject** i kontroluje do niego dostęp.

Celem takiego powiązania obiektów jest umożliwienie zastąpienia obiektu docelowego obiektem **Proxy**: klient, zamiast do obiektu docelowego, odwołuje się do obiektu **Proxy**, który deleguje żądania do niego lub próbuje obsługiwać je samodzielnie. W szczególności obiekt **Proxy** może utworzyć obiekt **RealSubject** znacznie później niż klient może korzystać z niego, a tym samym opóźnić inicjację tego obiektu. Pozwala to m.in. na oszczędność czasu i innych zasobów.

Zaawansowane projektowanie obiektowe

Proxy: uczestnicy


- **Proxy**
  - posiada referencję do obiektu *Real Subject* i deleguje do niego żądania
  - kontroluje dostęp do obiektu *Real Subject*
  - jest zamiennikiem *Real Subject* dla klienta
- **Subject**
  - definiuje wspólny interfejs dla *Proxy* i *Real Subject*
- **Real Subject**
  - rzeczywisty obiekt wymagający kontroli i ochrony

Wzorce projektowe cz. I (35)

Obiekt Proxy pełni główną rolę we wzorcu: zarządza podległym mu obiektem RealSubject i podejmuje decyzje dotyczące utworzenia go, przekazania mu sterowania etc. W ten sposób pełni funkcje ochronne (uniemożliwia nieautoryzowany dostęp) oraz kontrolne w stosunku do niego.

Subject definiuje wspólny interfejs, poprzez który odbywa się wymiana komunikatów między klientem a układem Proxy – RealSubject.

Zaawansowane projektowanie obiektowe

Proxy: konsekwencje

- Zdalny obiekt *Proxy* jest **lokalnym reprezentantem** obiektu znajdującego się **w innej przestrzeni adresowej**
- Wirtualny obiekt *Proxy* pełni rolę zamiennika dla **obiekту o dużych wymaganiach zasobowych** (np. pamięciowych)
- Ochronny obiekt *Proxy* odostępnia obiekt *Real Subject* tylko **uprawnionym obiektom**

Wzorce projektowe cz. I (36)


Istnieją trzy podstawowe rodzaje wzorca Proxy:

Zdalny obiekt Proxy (ang. *remote proxy*) służy do reprezentacji obiektu znajdującego się w innej przestrzeni adresowej, np. na innym komputerze. Dzięki temu dla lokalnych klientów wszystkie odwołania są pozornie lokalne. Proxy przejmuje wówczas odpowiedzialność za zdalne wywołania metod poprzez sieć, serializację parametrów i odebranie wyników. Mechanizm ten jest stosowany w większości środowisk przetwarzania rozproszonego np. CORBA lub EJB.

Wirtualny obiekt Proxy zastępuje obiekt RealSubject o dużych wymaganiach zasobowych, np. alokujący duży obszar pamięci. Aby opóźnić (a w szczególnych przypadkach nawet zastąpić) proces tworzenia takiego obiektu, Proxy obsługuje wszystkie zadania obiektu RealSubject, które nie wymagają odwołań do tego obszaru pamięci.

Ochronny obiekt Proxy zajmuje się zabezpieczeniem dostępu do obiektu RealSubject przed nieautoryzowanym dostępem. Obiekt RealSubject nigdy nie jest bezpośrednio dostępny dla klientów; w ich imieniu występuje Proxy, który określa, którym z nich można udostępnić usługi oferowane przez RealSubject, a którym nie.

Zaawansowane projektowanie obiektowe

Command: cel

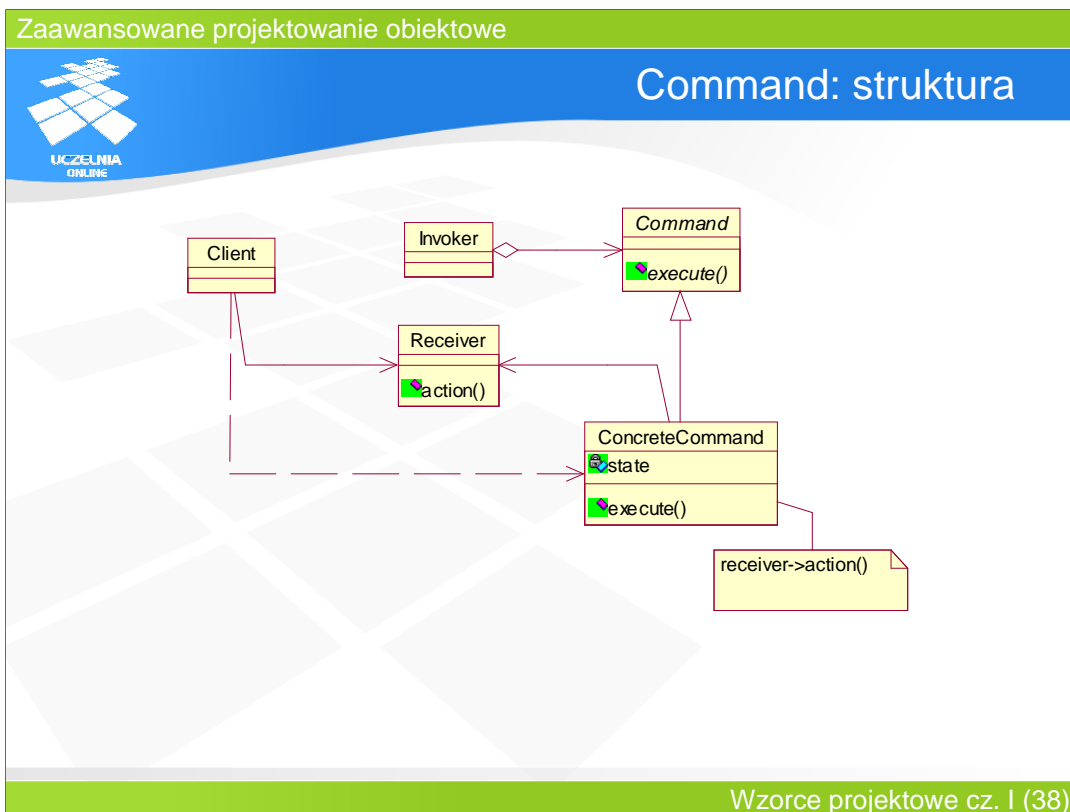
- Hermetyzacja poleceń do wykonania w postaci obiektów
- Umożliwienie parametryzacji klientów obiektami poleceń
- Wsparcie dla poleceń odwracalnych

*E. Gamma et al. (1995)*

Wzorce projektowe cz. I (37)

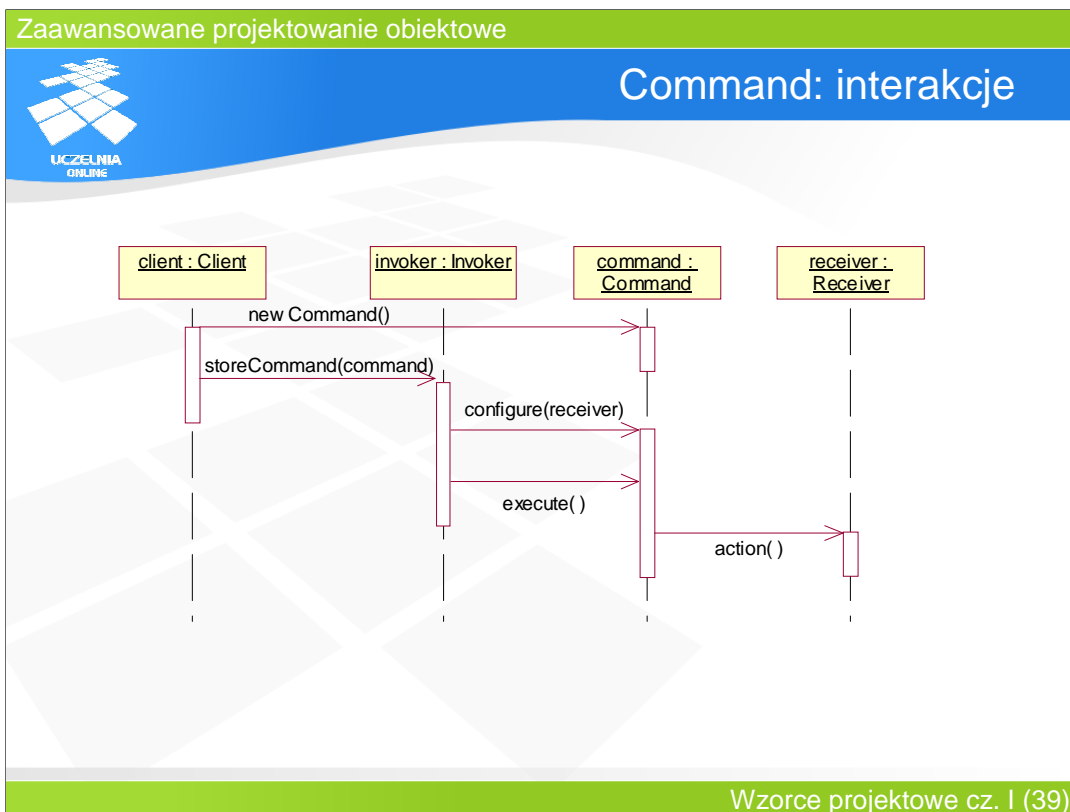
Wzorzec Command pozwala hermetyzować polecenia do wykonania w postaci obiektów, aby można było traktować je w sposób abstrakcyjny i np. przekazywać jako parametry. W języku C istnieje możliwość przekazania wskaźnika na funkcję. W wysokopoziomowych językach obiektowych, które tej możliwości nie posiadają, ten sam efekt można osiągnąć poprzez przekazanie referencji lub wskaźnika do obiektu definiującego określoną metodę.

Takie rozwiązanie zapewnia hermetyzację poleceń, możliwość abstrahowania od ich przeznaczenia, a przy okazji umożliwia stosowanie np. poleceń odwracalnych (o ile obiekt reprezentujący polecenie zapamiętuje stan sprzed jego wykonania).



Podstawowym elementem wzorca jest interfejs **Command**, deklarujący metodę `execute()`. Jest to polimorficzna metoda reprezentująca polecenie do wykonania. Metoda ta jest implementowana w klasach **ConcreteCommand** w postaci polecenia wykonania określonej akcji na obiekcie-przedmiocie **Receiver**.

Warto zauważyć, że klient nie jest bezpośrednio związany ani z obiektem **Command**, ani z obiektem inicjującym jego wywołanie, czyli **Invoker**. Widzi jedynie odbiorcę wyników operacji – obiekt **Receiver**.



Szczegółowy przepływ sterowania przedstawia diagram sekwencji. Inicjatorem przetwarzania jest obiekt *Invoker*, który zarządza obiektami typu *Command*. W momencie nadejścia żądania wykonania określonej operacji *Invoker* parametryzuje skojarzony z nią obiekt *Command* właściwym odbiorcą ich działań, czyli obiektem *Receiver*. Następnie wywołuje metodę *execute()* w tym obiekcie, powodując określone skutki w obiekcie *Receiver*, widoczne dla Klienta.



- **Command**
  - definiuje interfejs obiektu reprezentującego polecenie
- **Concrete Command**
  - jest powiązany z właściwym obiektem *Receiver*
  - implementuje akcję w postaci metody *execute()*
- **Client**
  - tworzy *Concrete Command*
- **Invoker**
  - ustala odbiorcę akcji każdego obiektu *Command*
  - wywołuje metodę *execute()* obiektu *Command*
- **Receiver**
  - jest przedmiotem akcji wykonanej przez *Command*

Role poszczególnych obiektów zostaną omówione na przykładzie. W aplikacji okienkowej polecenia znajdujące się w menu są zdefiniowane w postaci obiektów typu *Command*. Każde polecenie jest inną implementacją tego interfejsu, i posiada innego odbiorcę, ustalanego w momencie wykonywania akcji (np. polecenie zamknięcia okna działa na aktualnie aktywne okno). W momencie kliknięcia na wybranej pozycji menu (czyli obiektu *Invoker*), wykonuje ona metodę *execute()* skojarzonego z nią polecenia typu *Command*, ustalając jego odbiorcę. Efekt, w postaci np. zamknięcia okna, jest widoczny dla klienta.

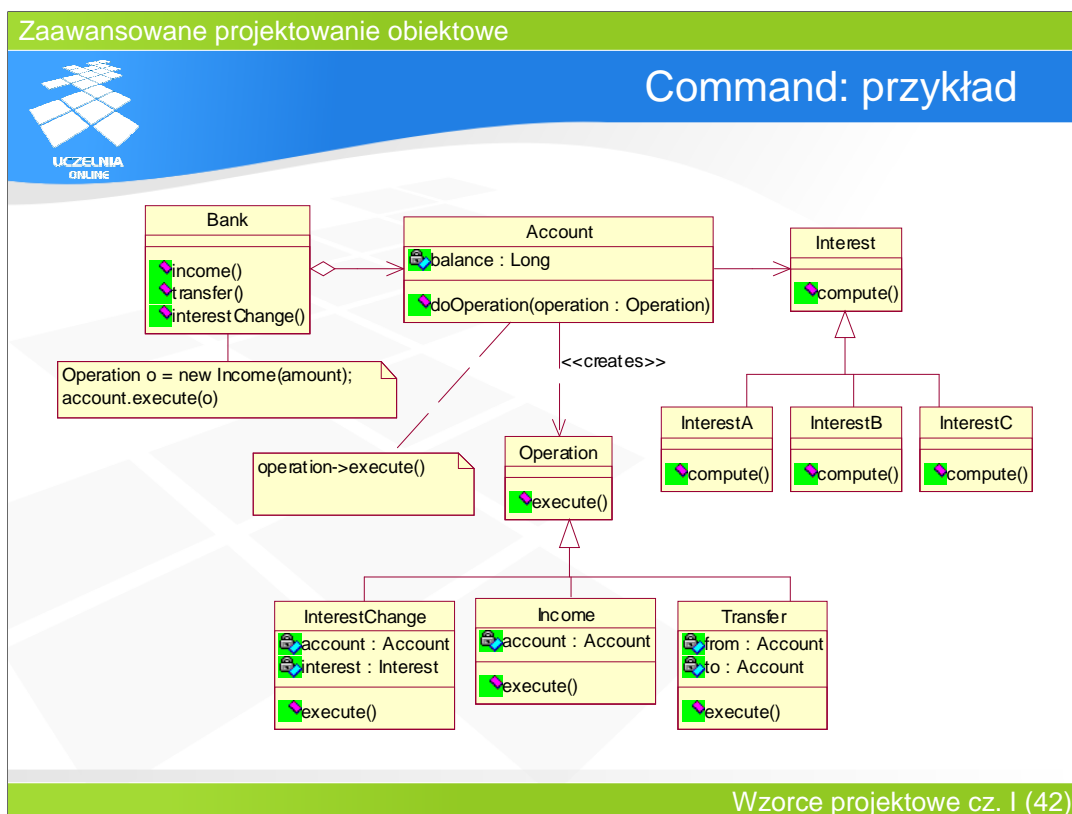




- Usunięcie powiązania między nadawcą i przedmiotem polecenia
- Łatwe dodawanie kolejnych obiektów *Command*
- Możliwość manipulacji obiektami *Command*
  - polecenia złożone: wzorzec *Composite*
- Polecenia mogą być odwracalne
  - zapamiętanie stanu przez *Concrete Command*
  - wykorzystanie wzorca *Memento*

Istotną korzyścią płynącą z zastosowania wzorca jest rozdzielenie zależności pomiędzy nadawcą (Klientem) i odbiorcą (obiektem Receiver) komunikatu. Zastosowanie polimorfizmu pozwala traktować poszczególne polecenia abstrakcyjnie, a co za tym idzie – dodawać nowe typy poleceń bez konieczności zmiany struktury systemu. Poszczególne obiekty Command mogą być dowolnie złożone, także w postaci kompozytów innych poleceń.

Dodatkową zaletą użycia obiektu do hermetyzacji poleceń jest możliwość utworzenia w typie Command przeciwstawnej metody, która odwraca efekt wykonania polecenia. W takiej sytuacji obiekt ConcreteCommand musi zapamiętać stan obiektu Receiver sprzed wykonania operacji lub np. skorzystać z wzorca Memento.



Bank zarządza grupą obiektów Account reprezentujących rachunki bankowe. Operacje bankowe, wykonywane na rachunkach, są implementacjami interfejsu Operation, posiadającego metodę *execute()*. Jej implementacja zależy od rodzaju operacji, dlatego w przypadku obiektu InterestChange będzie ona zmieniała stopę procentową, a w przypadku obiektu Transfer – dokonywała przelewu. Ponieważ każda operacja wymaga innych parametrów, dlatego są one przekazywane w konstruktorze poszczególnej klasy, a nie bezpośrednio w metodzie *execute()*. W tym przykładzie rolę obiektu Invoker pełni bank, ponieważ on wykonuje metodę *execute()*, a rolę przedmiotu polecenia (obektu Receiver) – obiekt Account.



```
public class Bank { // Invoker, Client
    public void income(Account acc, long amount) {
        Operation oper = new Income(amount);
        acc.doOperation(oper);
    }

    public void transfer(Account from, Account to, long amount){
        Operation oper = new Transfer(to, amount);
        from.doOperation(oper);
    }
}

public class Account { // Reciever
    long balance = 0;
    Interest interest = new InterestA();
    History history = new History();

    public void doOperation(Operation oper) {
        oper.execute(this);
        history.log(oper);
    }
}
```

Na slajdzie przedstawiono przykładową implementację klasy Bank, która pełni role Invoker i Client, oraz klasy Account, będącej odbiorcą poleceń. Klasa Bank definiuje metodę *income()*, która służy do wykonywania wpłaty na określony rachunek. W tym celu tworzy on instancję odpowiedniej operacji (klasy Income), a następnie przekazuje jej wykonanie obiektowi Account.

Klasa Account wykonuje dowolną abstrakcyjną operację przekazaną z zewnątrz, np. przez klasę Bank. Dzięki temu dodanie nowej operacji bankowej nie powoduje konieczności jakiegokolwiek zmiany w klasie Account.



```
abstract public class Operation { // Command
    public void execute();
}

public class Income { // ConcreteCommand1
    public Income(long amount) {
        // store parameters...
    }
    public void execute(Account acc) {
        acc.add(amount);
    }
}

public class Transfer { // ConcreteCommand2
    public Income(Account to, long amount) {
        // store parameters...
    }
    public void execute(Account from) {
        from.subtract(amount);
        to.add(amount);
    }
}
```

Klasa `Operation` pełni rolę obiektu `Command` we wzorcu i definiuje abstrakcyjną metodę `execute()`. Jest ona pokrywana w klasach reprezentujących poszczególne operacje bankowe, które implementują ją zgodnie ze specyfiką wykonywanej operacji.

Zaawansowane projektowanie obiektowe

UCZELNIA  
ONLINE

c.d.n.

Dalsza część katalogu wzorców projektowych zostanie przedstawiona na kolejnym wykładzie

Wzorce projektowe cz. I (45)

Kolejna część katalogu wzorców projektowych zostanie przedstawiona podczas kolejnego wykładu.